UNIVERSITY OF EXETER
Q-STEP CENTRE

Q-Step

A step-change in quantitative social science skills

Funded by the Nuffield Foundation, ESRC and HEFCE

# A Practical Introduction to Python

Part 1

Lewys Brace
l.brace@Exeter.ac.uk

# An introduction to coding with Python

An introductory coding course with Python…

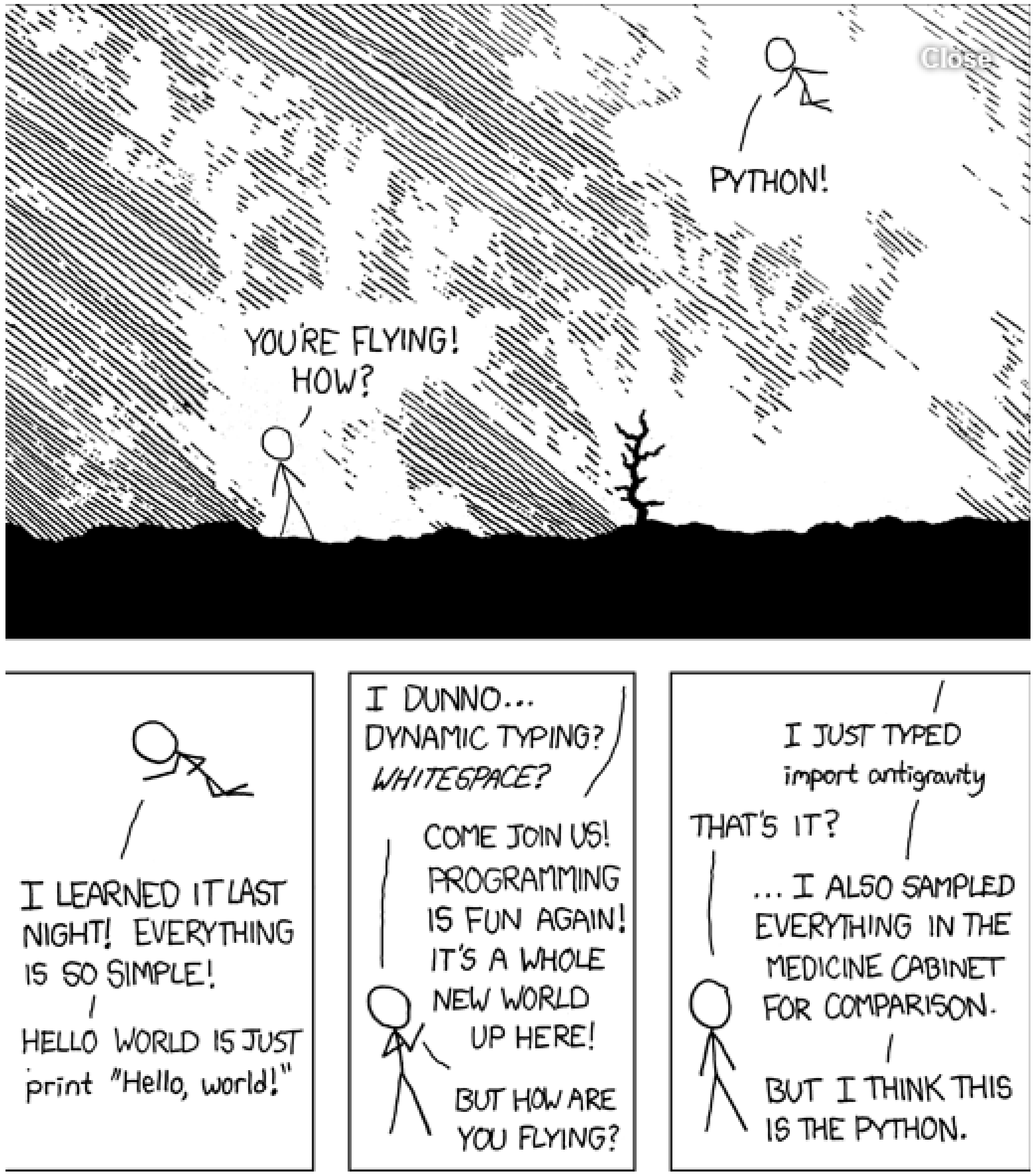
# Interpretive vs compiled languages

- Python is an interpretive language.
- This means that your code is not directly run by the hardware. It is instead passed to a *virtual machine*, which is just another programme that reads and interprets your code. If your code used the ‘+’ operation, this would be recognised by the interpreter at run time, which would then call its own internal function ‘add(a,b)’, which would then execute the machine code ‘ADD’.
- This is in contrast to compiled languages, where your code is translated into native machine instructions, which are then directly executed by the hardware. Here, the ‘+’ in your code would be translated directly in the ‘ADD’ machine code.

# Advantages of Python?

Because Python is an interpretive language, it has a number of advantages:

- Automatic memory management.
- Expressivity and syntax that is 'English'.
- Ease of programming.
- Minimises development time.
- Python also has a focus on *importing* modules, a feature that makes it useful for scientific computing.

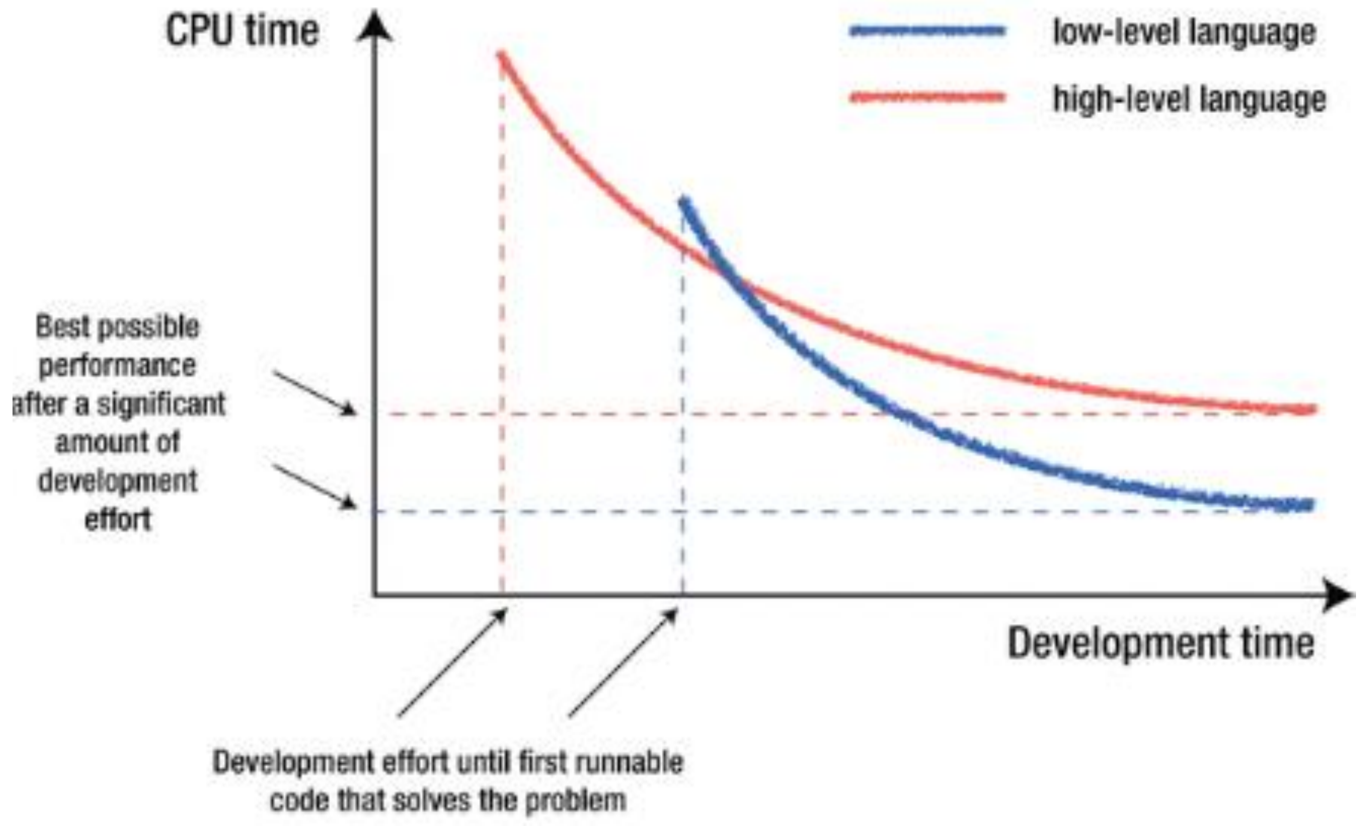

Close
PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

## Disadvantages

- Interpreted languages are slower than compiled languages.
- The modules that you import are developed in a decentralised manner; this can cause issues based upon individual assumptions.
- Multi-threading is hard in Python

# Which language is the best

- No one language is better than all others.
- The ‘best’ language depends on the task you are using it for and your personal preference.

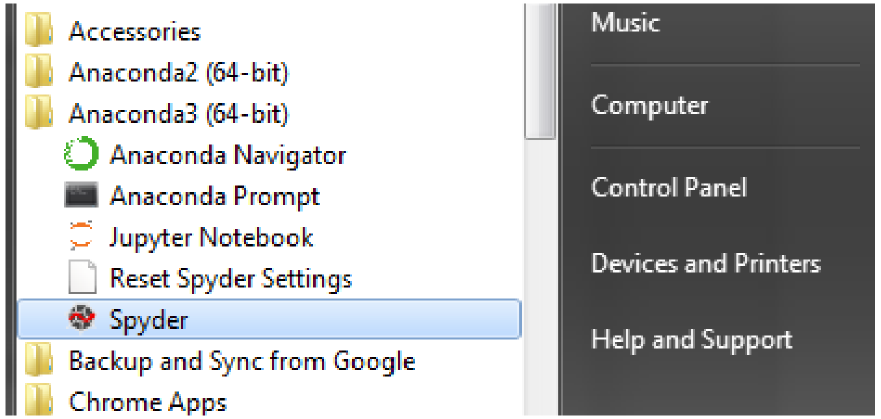


## Versions of Python

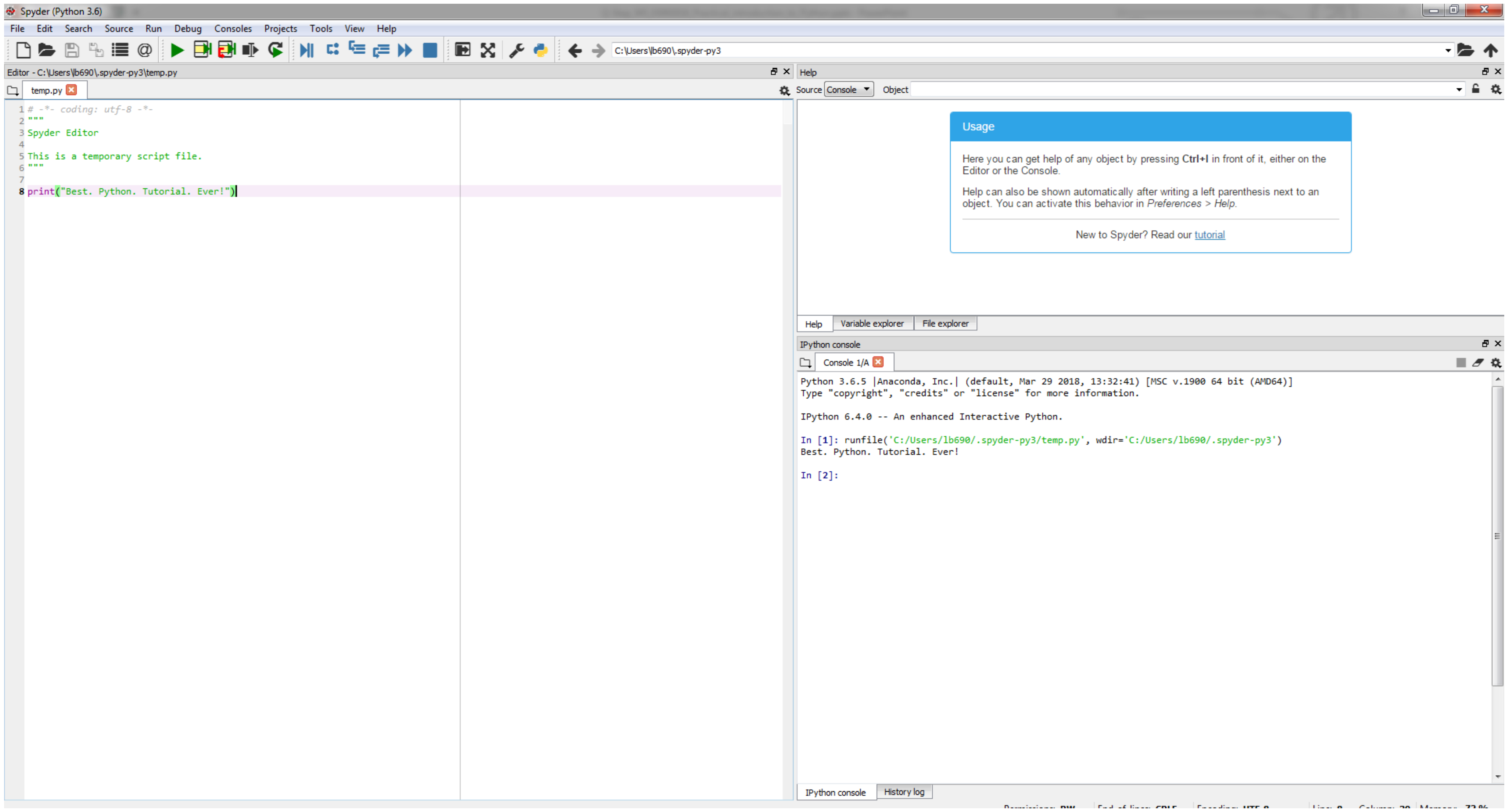
- There are currently two versions of Python in use; Python 2 and Python 3.
- Python 3 is not backward compatible with Python 2.
- A lot of the imported modules were only available in Python 2 for quite some time, leading to a slow adoption of Python 3. However, this not really an issue anymore.
- Support for Python 2 will end in 2020.

## The Anaconda IDE…

- The Anaconda distribution is the most popular Python distribution out there.
- Most importable packages are pre-installed.
- Offers a nice GUI in the form of Spyder.
- Before we go any further, let's open Spyder:



SPYDER


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\lb690\.spyder-py3
Editor - C:\Users\lb690\.spyder-py3\temp.py
temp.py
Help
Source Console Object
Help Variable explorer File explorer
IPython console
Console 1/A
IPython console History log

```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""

print("Best. Python. Tutorial. Ever!")
```

**Usage**

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

New to Spyder? Read our tutorial

```
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.4.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/lb690/.spyder-py3/temp.py', wdir='C:/Users/lb690/.spyder-py3')
Best. Python. Tutorial. Ever!

In [2]:
```

## Variables

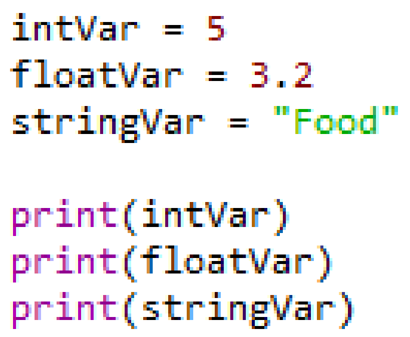
- Variables in python can contain alphanumerical characters and some special characters.
- By convention, it is common to have variable names that start with lower case letters and have class names beginning with a capital letter; but you can do whatever you want.
- Some keywords are reserved and cannot be used as variable names due to them serving an in-built Python function; i.e. `and`, `continue`, `break`. Your IDE will let you know if you try to use one of these.
- Python is dynamically typed; the type of the variable is derived from the value it is assigned.

# Variable types

- Integer (int)
- Float (float)
- String (str)
- Boolean (bool)
- Complex (complex)
- [...]
- User defined (classes)

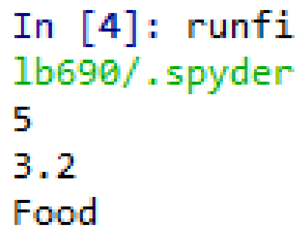
- A variable is assigned using the = operator; i.e:

In:



```
intVar = 5
floatVar = 3.2
stringVar = "Food"

print(intVar)
print(floatVar)
print(stringVar)
```

Out:

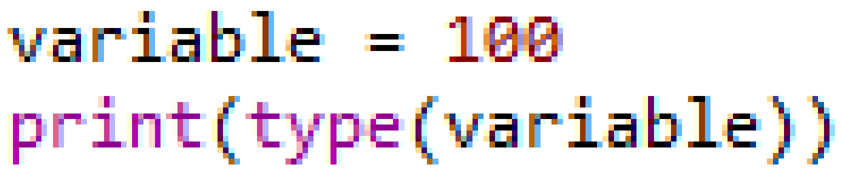


```
In [4]: runfi
1b690/.spyder
5
3.2
Food
```

- The print() function is used to print something to the screen.
- Create an integer, float, and string variable.
- Print these to the screen.
- Play around using different variable names, etc.

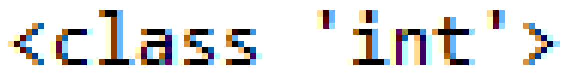
- You can always check the type of a variable using the type() function.

In:



```
variable = 100
print(type(variable))
```

Out:

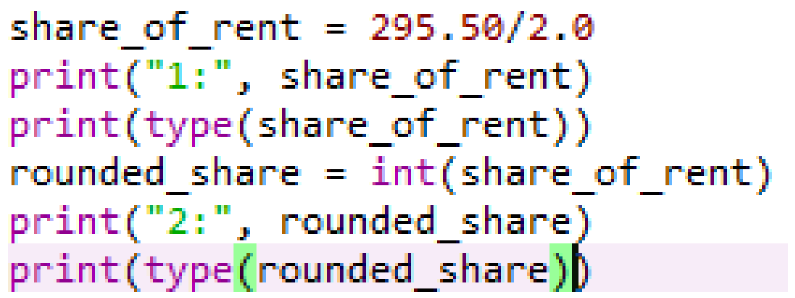


```
<class 'int'>
```

- Check the type of one of your variables.

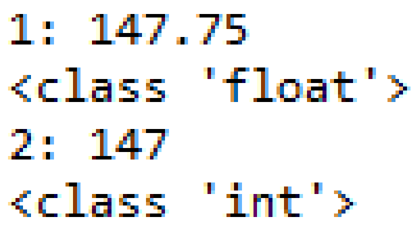
- Variables can be *cast* to a different type.

In:



```python
share_of_rent = 295.50/2.0
print("1:", share_of_rent)
print(type(share_of_rent))
rounded_share = int(share_of_rent)
print("2:", rounded_share)
print(type(rounded_share))
```

Out:



```
1: 147.75
<class 'float'>
2: 147
<class 'int'>
```

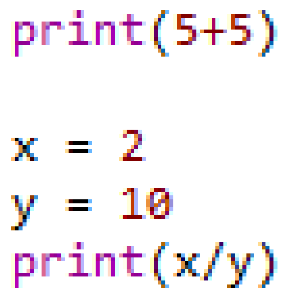
## Arithmetic operators

The arithmetic operators:

- Addition: +
- Subtract: -
- Multiplication: *
- Division: /
- Power: **

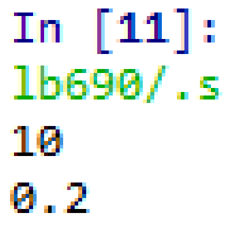
- Write a couple of operations using the arithmetic operators, and print the results to the screen.

In:



```
print(5+5)

x = 2
y = 10
print(x/y)
```

Out:



```
In [11]:
1b690/.s
10
0.2
```

# A quick note on the increment operator shorthand

- Python has a common idiom that is not necessary, but which is used frequently and is therefore worth noting:

```
x += 1
```

Is the same as:

```
x = x + 1
```

- This also works for other operators:

```
x += y          # adds y to the value of x
x *= y          # multiplies x by the value y
x -= y          # subtracts y from x
x /= y          # divides x by y
```

## Boolean operators

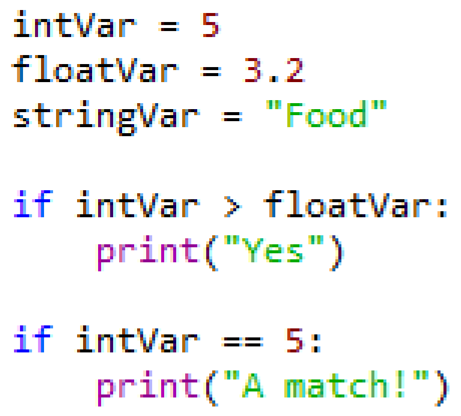
- Boolean operators are useful when making conditional statements, we will cover these in-depth later.
- and
- or
- not

## Comparison operators

- Greater than: >
- Lesser than: <
- Greater than or equal to: >=
- Lesser than or equal to: <=
- Is equal to: ==

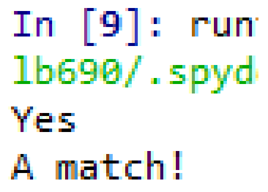
- Write a couple of operations using comparison operators; i.e.

In:



```
intVar = 5
floatVar = 3.2
stringVar = "Food"

if intVar > floatVar:
    print("Yes")

if intVar == 5:
    print("A match!")
```

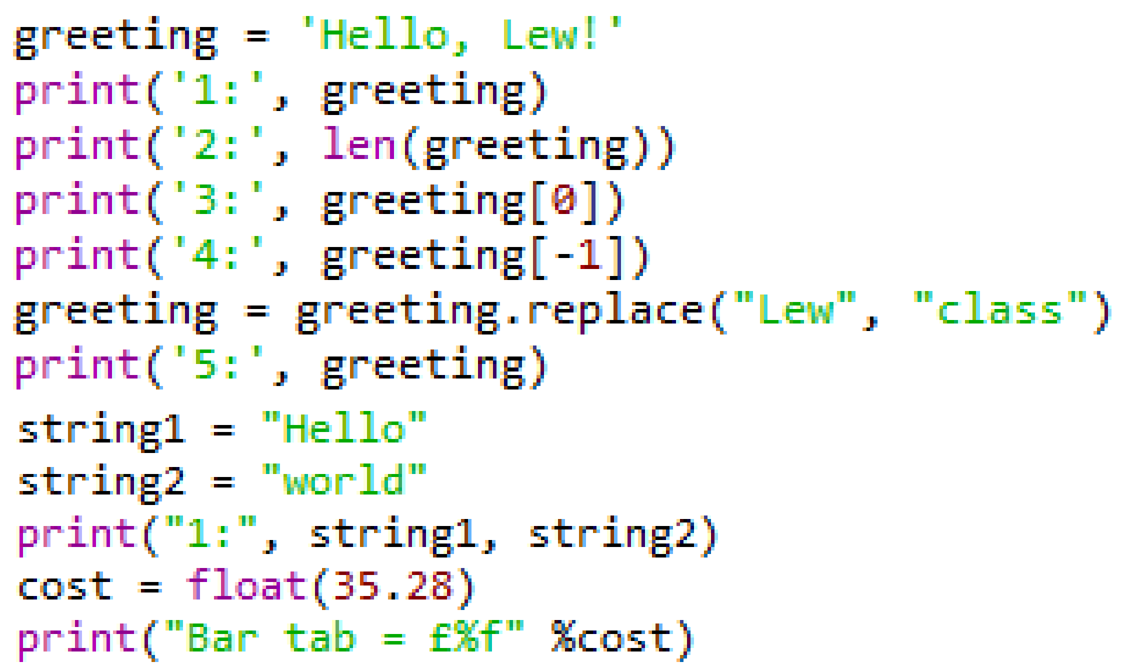
Out:



```
In [9]: run
1b690/.spyd
Yes
A match!
```

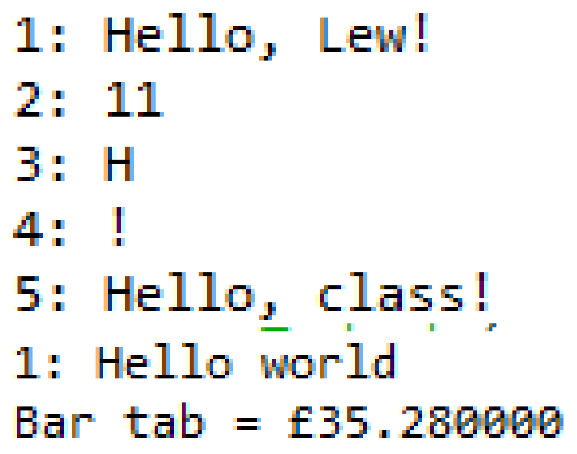
# Working with strings

In:



```
greeting = 'Hello, Lew!'
print('1:', greeting)
print('2:', len(greeting))
print('3:', greeting[0])
print('4:', greeting[-1])
greeting = greeting.replace("Lew", "class")
print('5:', greeting)
string1 = "Hello"
string2 = "world"
print("1:", string1, string2)
cost = float(35.28)
print("Bar tab = £%f" %cost)
```

Out:



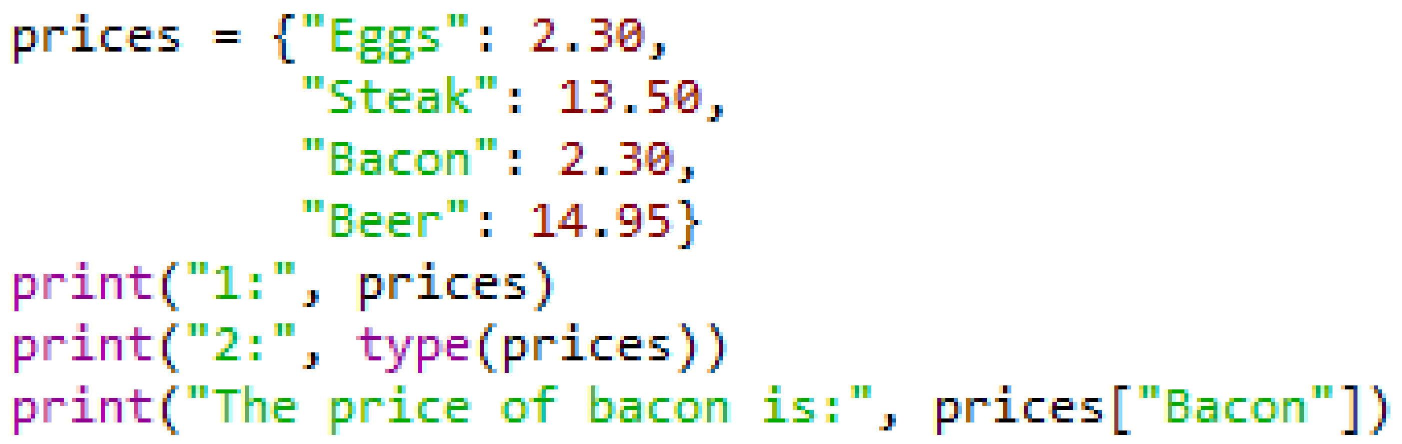
```
1: Hello, Lew!
2: 11
3: H
4: !
5: Hello, class!
1: Hello world
Bar tab = £35.280000
```

- Create a string variable.
- Work out the length of the string.

## Dictionaries

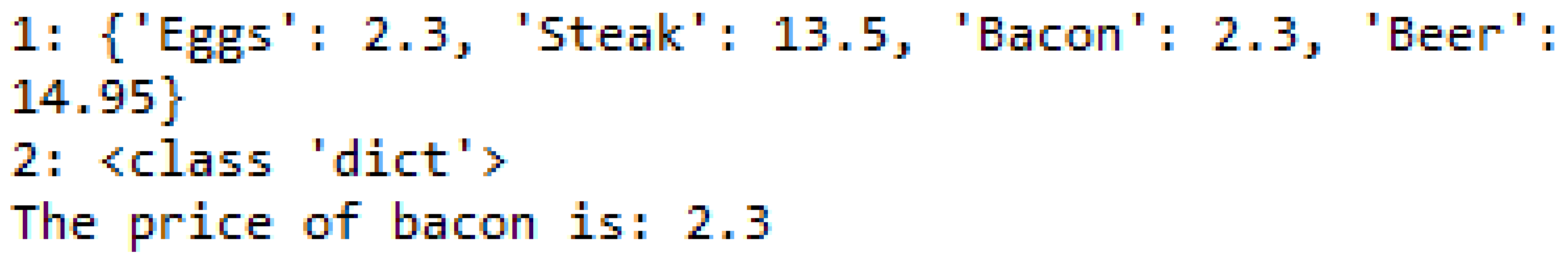
- Dictionaries are lists of key-valued pairs.

In:



```python
prices = {"Eggs": 2.30,
          "Steak": 13.50,
          "Bacon": 2.30,
          "Beer": 14.95}
print("1:", prices)
print("2:", type(prices))
print("The price of bacon is:", prices["Bacon"])
```

Out:

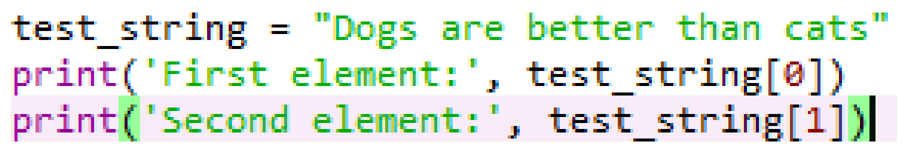


```
1: {'Eggs': 2.3, 'Steak': 13.5, 'Bacon': 2.3, 'Beer':
14.95}
2: <class 'dict'>
The price of bacon is: 2.3
```

# Indexing

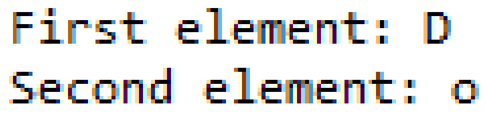
- Indexing in Python is 0-based, meaning that the first element in a string, list, array, etc, has an index of 0. The second element then has an index of 1, and so on.

In:



```python
test_string = "Dogs are better than cats"
print('First element:', test_string[0])
print('Second element:', test_string[1])
```

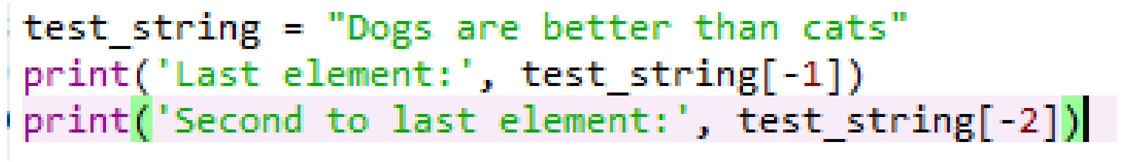
Out:



```
First element: D
Second element: o
```

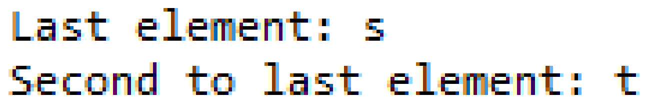
- You can cycle backwards through a list, string, array, etc, by placing a minus symbol in front of the index location.

In:



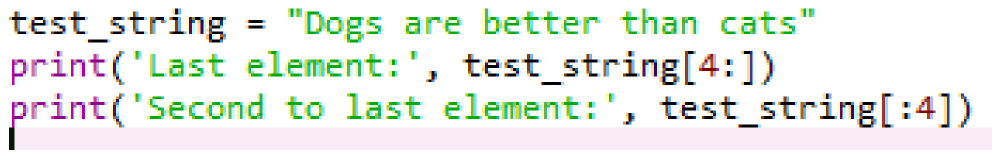
```python
test_string = "Dogs are better than cats"
print('Last element:', test_string[-1])
print('Second to last element:', test_string[-2])
```

Out:



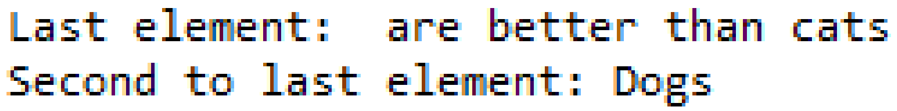
```
Last element: s
Second to last element: t
```

In:



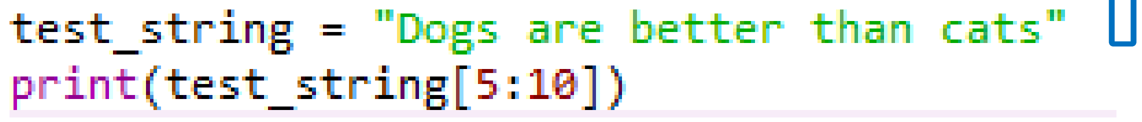
```python
test_string = "Dogs are better than cats"
print('Last element:', test_string[4:])
print('Second to last element:', test_string[:4])
```

Out:



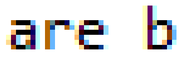
```
Last element:  are better than cats
Second to last element: Dogs
```

In:



```python
test_string = "Dogs are better than cats"
print(test_string[5:10])
```

Out:

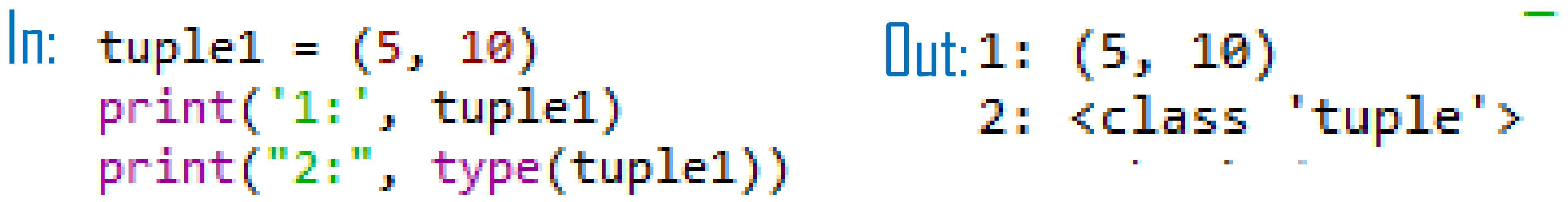


```
are b
```

- Create a string that is 10 characters in length.
- Print the second character to the screen.
- Print the third to last character to the screen.
- Print all characters after the fourth character.
- Print characters 2-8.

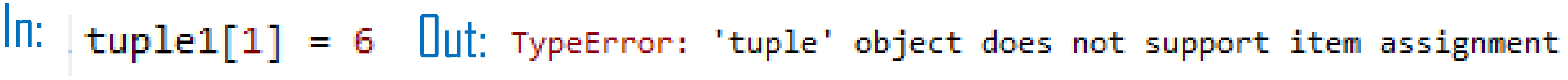
# Tuples

- Tuples are containers that are immutable; i.e. their contents cannot be altered once created.



```
In: tuple1 = (5, 10)
    print('1:', tuple1)
    print("2:", type(tuple1))
```

```
Out:1: (5, 10)
    2: <class 'tuple'>
```



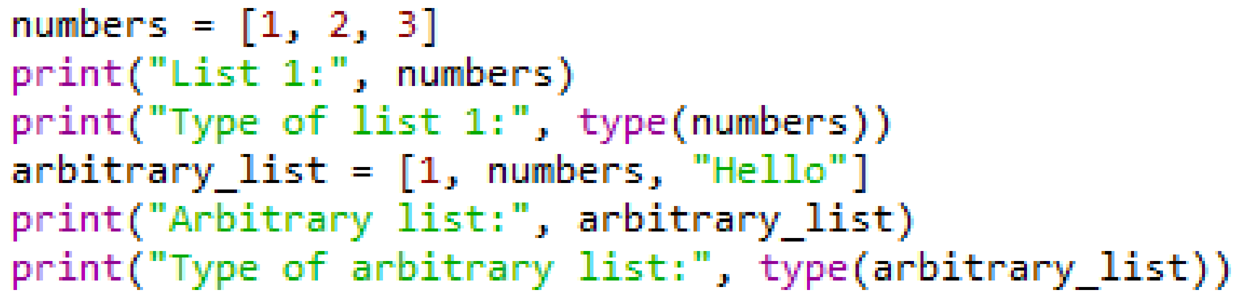
```
In: tuple1[1] = 6
```

```
Out: TypeError: 'tuple' object does not support item assignment
```

# Lists

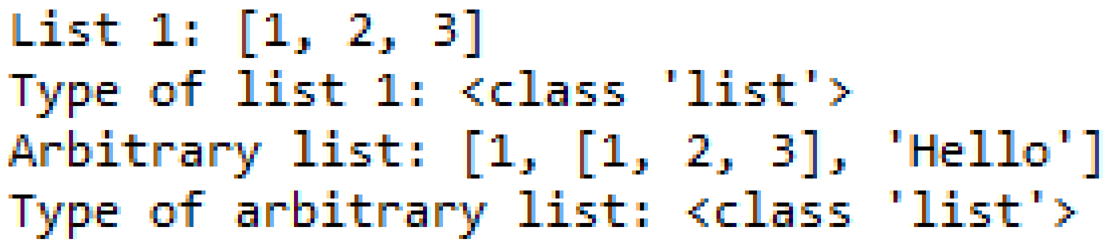
- Lists are essentially containers of arbitrary type.
- They are probably the container that you will use most frequently.
- The elements of a list can be of different types.
- The difference between tuples and lists is in performance; it is much faster to ‘grab’ an element stored in a tuple, but lists are much more versatile.
- Note that lists are denoted by [] and not the () used by tuples.

In:



```
numbers = [1, 2, 3]
print("List 1:", numbers)
print("Type of list 1:", type(numbers))
arbitrary_list = [1, numbers, "Hello"]
print("Arbitrary list:", arbitrary_list)
print("Type of arbitrary list:", type(arbitrary_list))
```

Out:



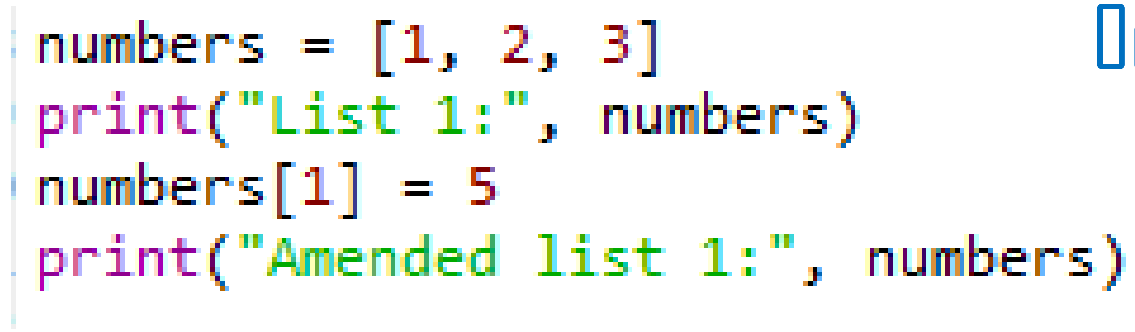
```
List 1: [1, 2, 3]
Type of list 1: <class 'list'>
Arbitrary list: [1, [1, 2, 3], 'Hello']
Type of arbitrary list: <class 'list'>
```

- Create a list and populate it with some elements.

## Adding elements to a list

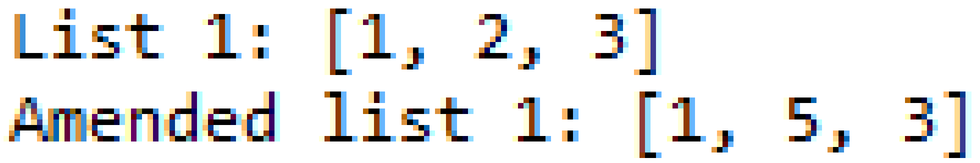
- Lists are mutable; i.e. their contents can be changed. This can be done in a number of ways.
- With the use of an index to replace a current element with a new one.

In:



```
numbers = [1, 2, 3]
print("List 1:", numbers)
numbers[1] = 5
print("Amended list 1:", numbers)
```

Out:

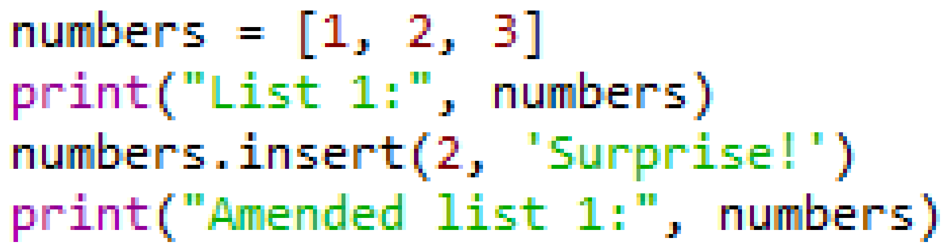


```
List 1: [1, 2, 3]
Amended list 1: [1, 5, 3]
```

- Replace the second element in your string with the integer 2.

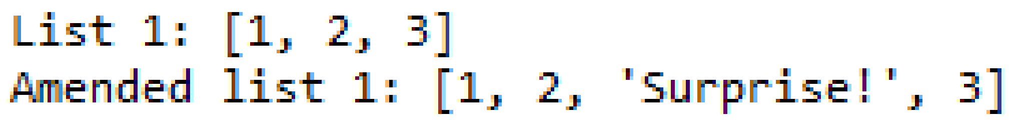
- You can use the insert() function in order to add an element to a list at a specific indexed location, without overwriting any of the original elements.

In:



```
numbers = [1, 2, 3]
print("List 1:", numbers)
numbers.insert(2, 'Surprise!')
print("Amended list 1:", numbers)
```

Out:

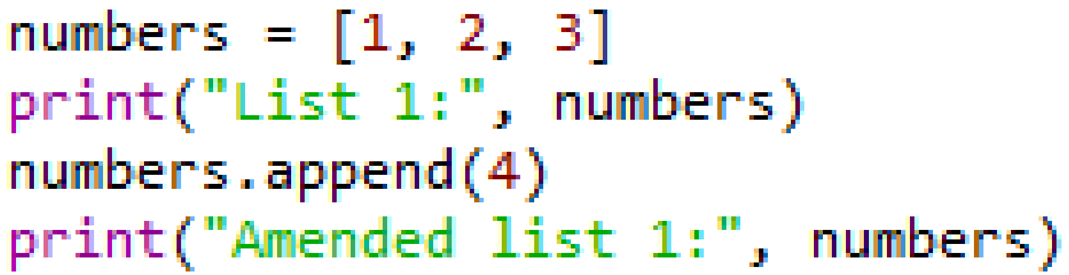


```
List 1: [1, 2, 3]
Amended list 1: [1, 2, 'Surprise!', 3]
```

- Use insert() to put the integer 3 after the 2 that you just added to your string.

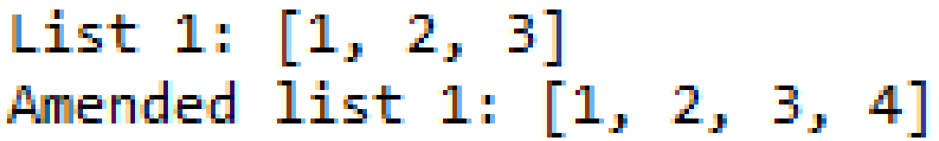
- You can add an element to the end of a list using the append() function.

In:



```
numbers = [1, 2, 3]
print("List 1:", numbers)
numbers.append(4)
print("Amended list 1:", numbers)
```

Out:



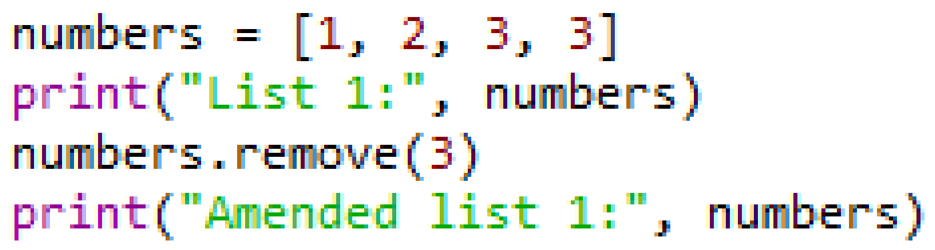
```
List 1: [1, 2, 3]
Amended list 1: [1, 2, 3, 4]
```

- Use append() to add the string “end” as the last element in your list.

# Removing elements from a list

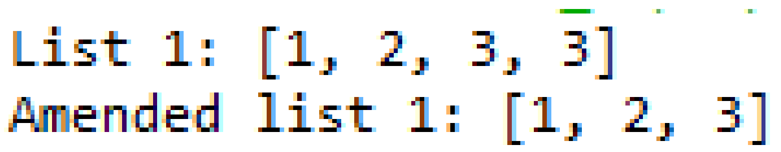
- You can remove an element from a list based upon the element value.
- Remember: If there is more than one element with this value, only the first occurrence will be removed.

In:



```
numbers = [1, 2, 3, 3]
print("List 1:", numbers)
numbers.remove(3)
print("Amended list 1:", numbers)
```

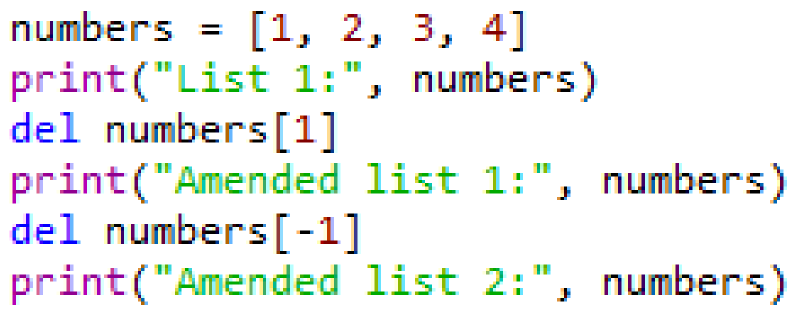
Out:



```
List 1: [1, 2, 3, 3]
Amended list 1: [1, 2, 3]
```

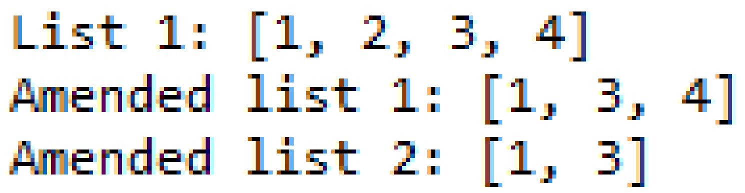
- It is better practice to remove elements by their index using the del function.

In:



```
numbers = [1, 2, 3, 4]
print("List 1:", numbers)
del numbers[1]
print("Amended list 1:", numbers)
del numbers[-1]
print("Amended list 2:", numbers)
```

Out:



```
List 1: [1, 2, 3, 4]
Amended list 1: [1, 3, 4]
Amended list 2: [1, 3]
```

- Use del to remove the 3 that you added to the list earlier.

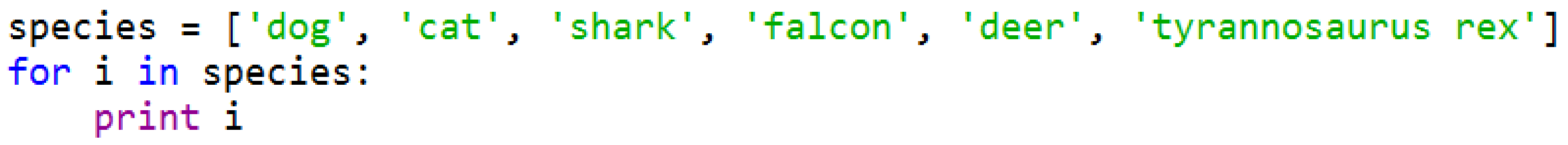
## For loops

- The for loop is used to iterate over elements in a sequence, and is often used when you have a piece of code that you want to repeat a number of times.
- For loops essentially say:

*“For all elements in a sequence, do something”*

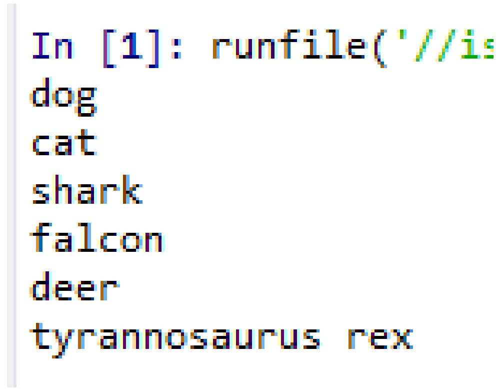
# An example

- We have a list of species:



```python
species = ['dog', 'cat', 'shark', 'falcon', 'deer', 'tyrannosaurus rex']
for i in species:
    print i
```

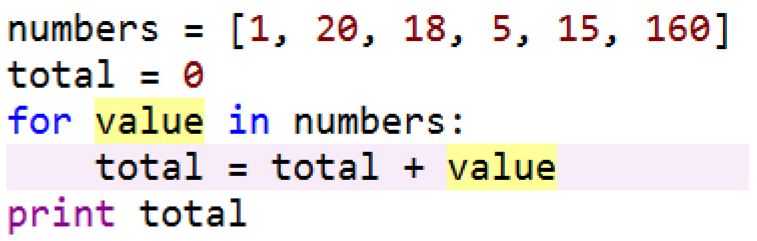
- The command underneath the list then cycles through each entry in the species list and prints the animal's name to the screen. Note: The i is quite arbitrary. You could just as easily replace it with 'animal', 't', or anything else.



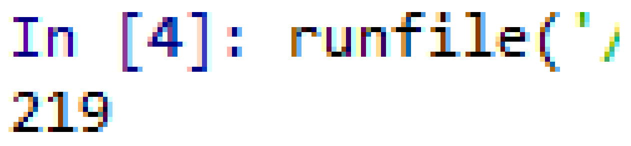
```
In [1]: runfile('//is
dog
cat
shark
falcon
deer
tyrannosaurus rex
```

## Another example

- We can also use for loops for operations other than printing to a screen. For example:



```
numbers = [1, 20, 18, 5, 15, 160]
total = 0
for value in numbers:
    total = total + value
print total
```



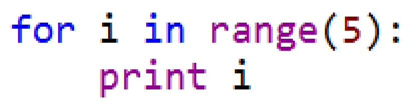
```
In [4]: runfile('/
219
```

- Using the list you made a moment ago, use a for loop to print each element of the list to the screen in turn.

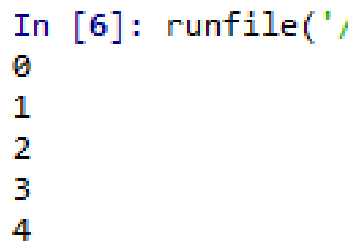
# The range() function

- The range() function generates a list of numbers, which is generally used to iterate over within for loops.
- The range() function has two sets of parameters to follow:

range(stop)
stop: Number of integers (whole numbers) to generate, starting from zero. i.e:

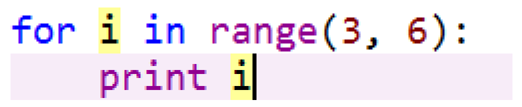


```
for i in range(5):
    print i
```

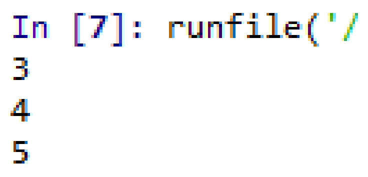


```
In [6]: runfile('/
0
1
2
3
4
```

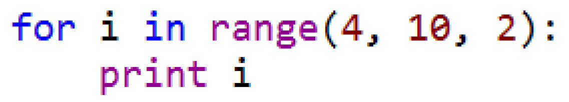
range([start], stop[, step])
start: Starting number of the sequence.
stop: Generate numbers up to, but not including this number.
step: Difference between each number in the sequence
i.e.:



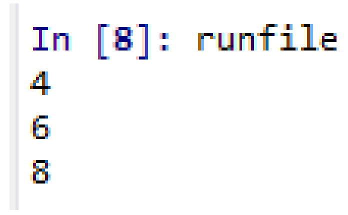
```
for i in range(3, 6):
    print i
```



```
In [7]: runfile('/
3
4
5
```



```
for i in range(4, 10, 2):
    print i
```



```
In [8]: runfile
4
6
8
```

**Note:**

- All parameters must be integers.
- Parameters can be positive or negative.
- The range() function (and Python in general) is 0-index based, meaning list indexes start at 0, not 1. eg. The syntax to access the first element of a list is mylist[0]. Therefore the last integer generated by range() is up to, but not including, stop.

- Create an empty list.

```
new_list = []
```

- Use the range() and append() functions to add the integers 1-20 to the empty list.

```
for i in range(1, 21):
    new_list.append(i)
```

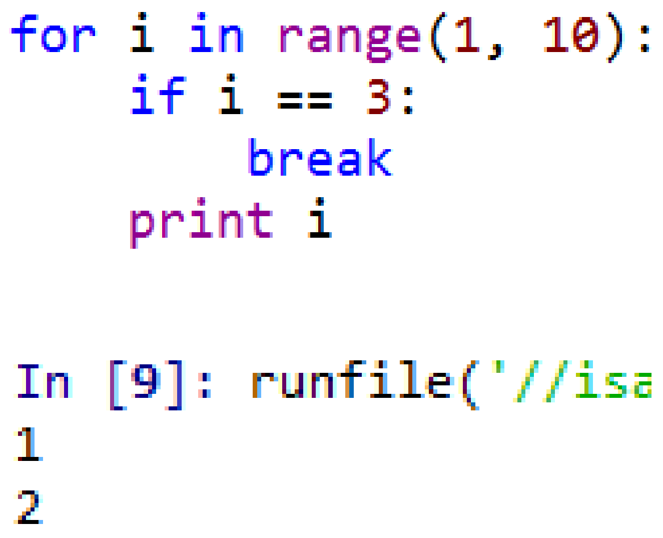
- Print the list to the screen, what do you have?

```
print(new_list)
```

Output:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

# The break() function

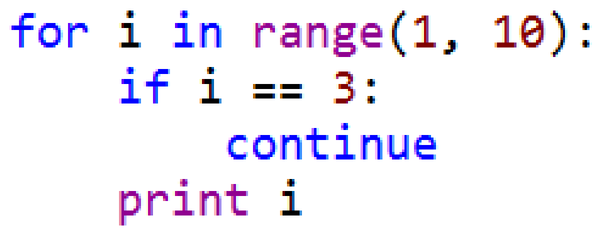
- To terminate a loop, you can use the break() function.
- The break() statement breaks out of the innermost enclosing for or while loop.



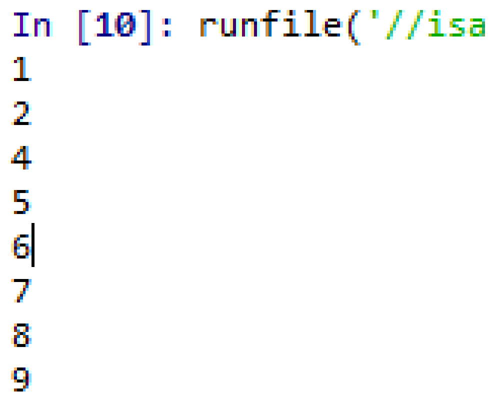
```
for i in range(1, 10):
    if i == 3:
        break
    print i


In [9]: runfile('//isa
1
2
```

# The continue () function

- The continue() statement is used to tell Python to skip the rest of the statements in the current loop block, and then to continue to the next iteration of the loop.



```python
for i in range(1, 10):
    if i == 3:
        continue
    print i
```



```
In [10]: runfile('//isa
1
2
4
5
6
7
8
9
```

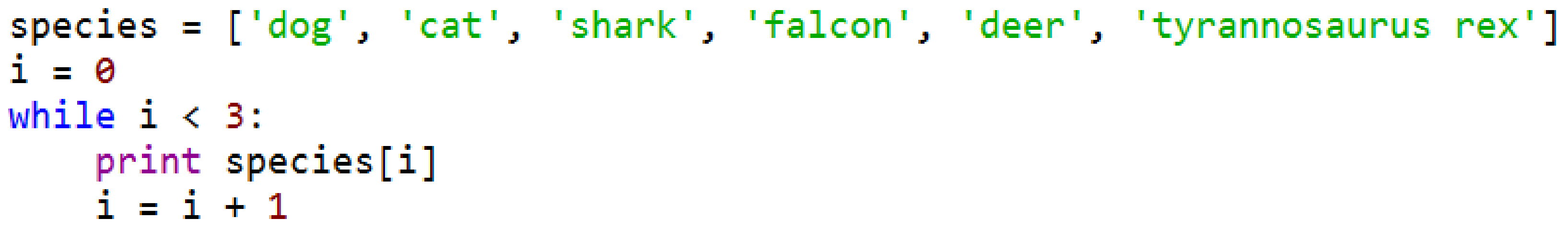
## While loops

- The while loop tells the computer to do something as long as a specific condition is met.
- It essentially says:

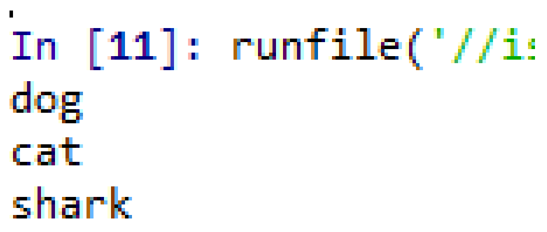
*“while this is true, do this.”*

- When working with while loops, its important to remember the nature of various operators.
- While loops use the `break()` and `continue()` functions in the same way as a for loop does.

# An example



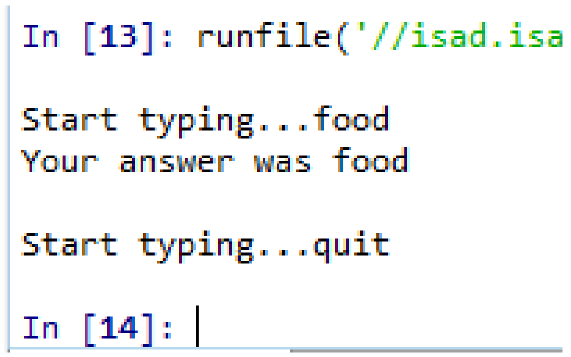
```
species = ['dog', 'cat', 'shark', 'falcon', 'deer', 'tyrannosaurus rex']
i = 0
while i < 3:
    print species[i]
    i = i + 1
```



```
In [11]: runfile('//i
dog
cat
shark
```

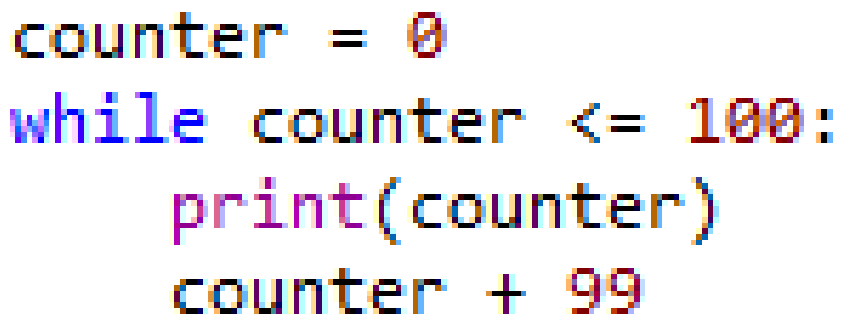
# Another example

```
while True:
    answer = raw_input("Start typing...")
    if answer == "quit":
        break
    print "Your answer was", answer
```



```
In [13]: runfile('//isad.isa

Start typing...food
Your answer was food

Start typing...quit

In [14]: 
```

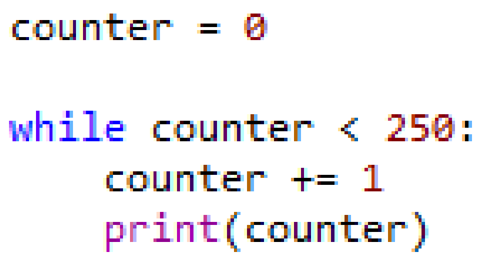
## A bad example



```
counter = 0
while counter <= 100:
    print(counter)
    counter + 99
```

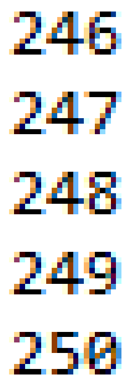
- Create a variable and set it to zero.
- Write a while loop that states that, while the variable is less than 250, add 1 to the variable and print the variable to the screen.

In:



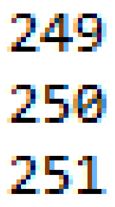
```
counter = 0

while counter < 250:
    counter += 1
    print(counter)
```

Out:



```
246
247
248
249
250
```

- Replace the < with <=, what happens?



```
249
250
251
```

## For loop vs. while loop

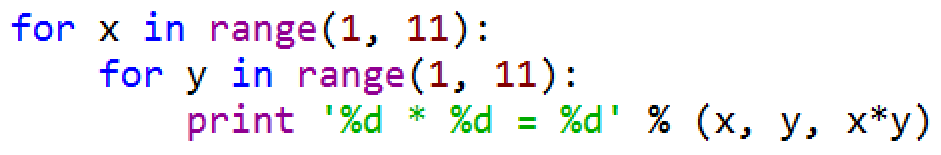
- You will use for loops more often than while loops.
- The for loop is the natural choice for cycling through a list, characters in a string, etc; basically, anything of *determinate* size.
- The while loop is the natural choice if you are cycling through something, such as a sequence of numbers, an *indeterminate* number of times until some condition is met.

## Nested loops

- In some situations, you may want a loop within a loop; this is known as a nested loop.

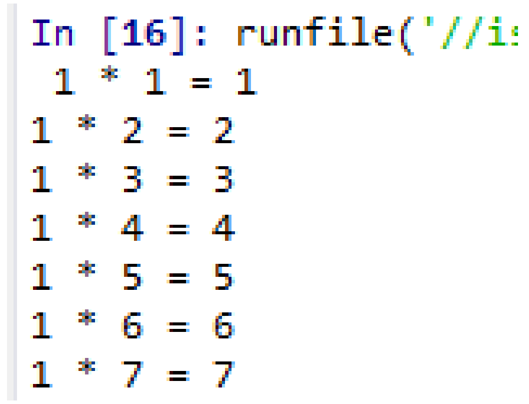
- What will the code on the right produce?
- Recreate this code and run it, what do you get?

In:



```
for x in range(1, 11):
    for y in range(1, 11):
        print '%d * %d = %d' % (x, y, x*y)
```

Out:

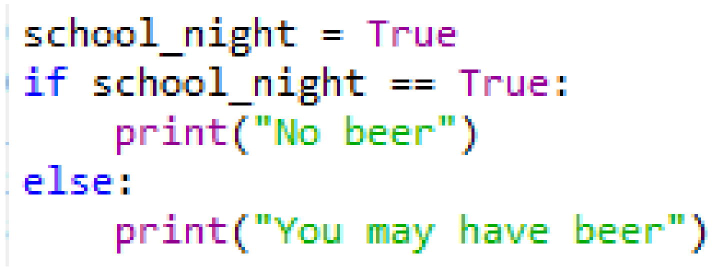


```
In [16]: runfile('//i
 1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
```

# Conditionals

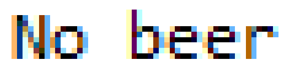
- There are three main conditional statements in Python; if, else, elif.
- We have already used if when looking at while loops.

In:



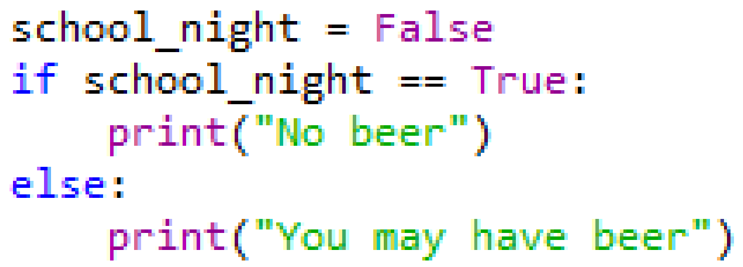
```python
school_night = True
if school_night == True:
    print("No beer")
else:
    print("You may have beer")
```

Out:



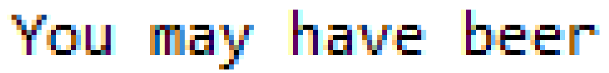
```
No beer
```

In:



```python
school_night = False
if school_night == True:
    print("No beer")
else:
    print("You may have beer")
```

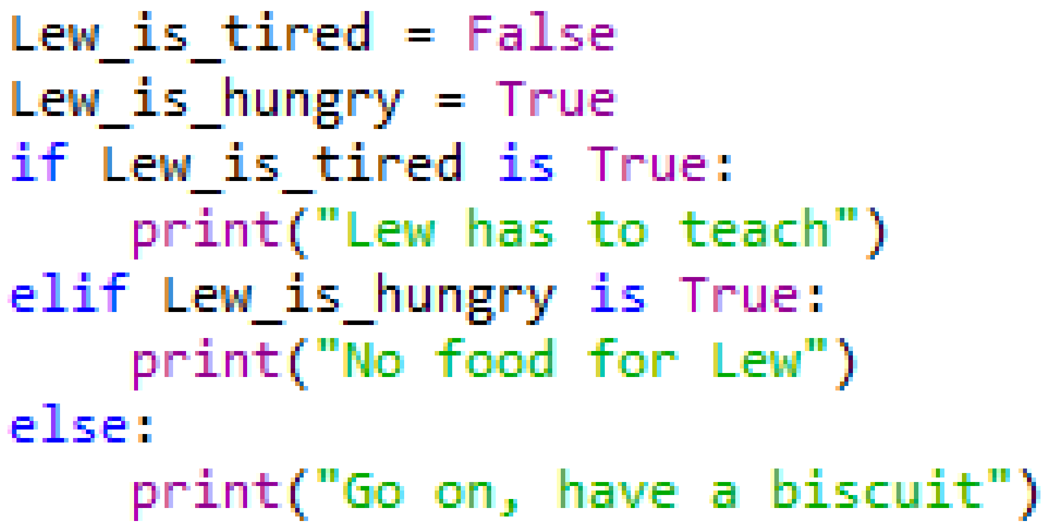
Out:



```
You may have beer
```

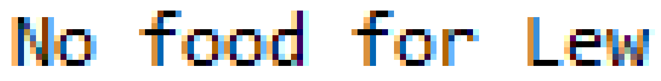
# An example of elif

In:



```python
Lew_is_tired = False
Lew_is_hungry = True
if Lew_is_tired is True:
    print("Lew has to teach")
elif Lew_is_hungry is True:
    print("No food for Lew")
else:
    print("Go on, have a biscuit")
```

Out:

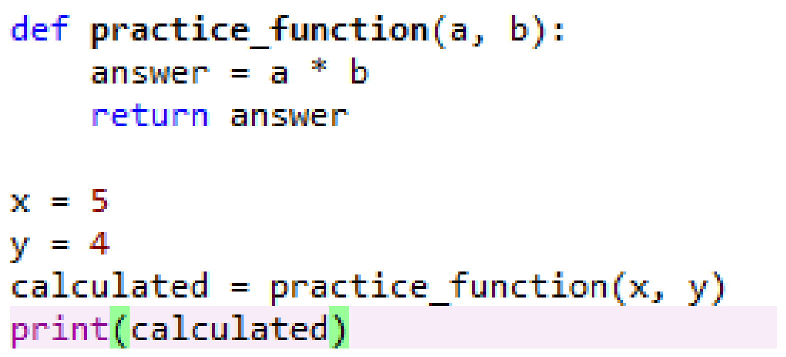


```
No food for Lew
```

# Functions

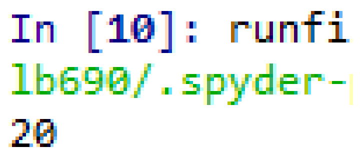
- A function is a block of code which only runs when it is called.
- They are really useful if you have operations that need to be done repeatedly; i.e. calculations.
- The function must be defined before it is called. In other words, the block of code that makes up the function must come before the block of code that makes use of the function.

In:



```
def practice_function(a, b):
    answer = a * b
    return answer

x = 5
y = 4
calculated = practice_function(x, y)
print(calculated)
```

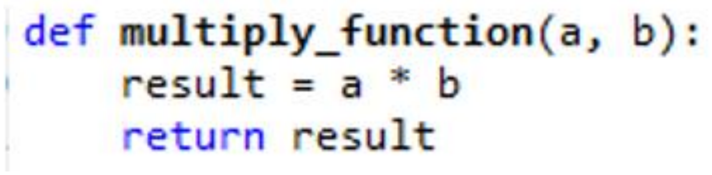
Out:



```
In [10]: runfi
1b690/.spyder-
20
```

- Create a function that takes two inputs, multiplies them, and then returns the result. It should look some like:

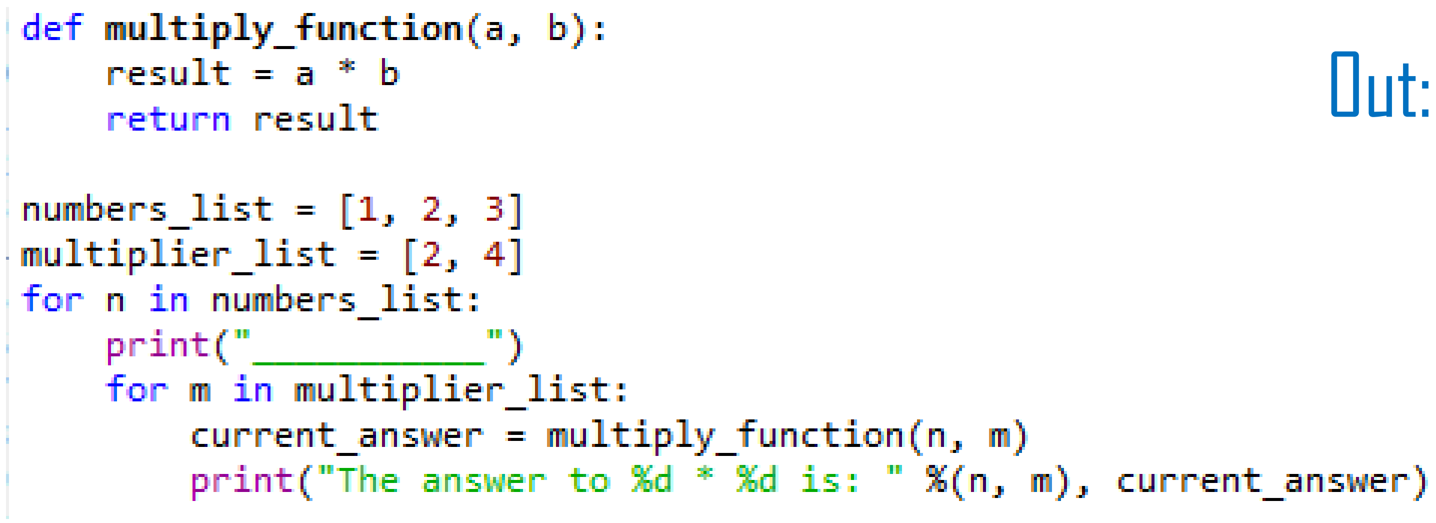
```
def function_name(a, b):
        do something
        return something
```



```
def multiply_function(a, b):
    result = a * b
    return result
```

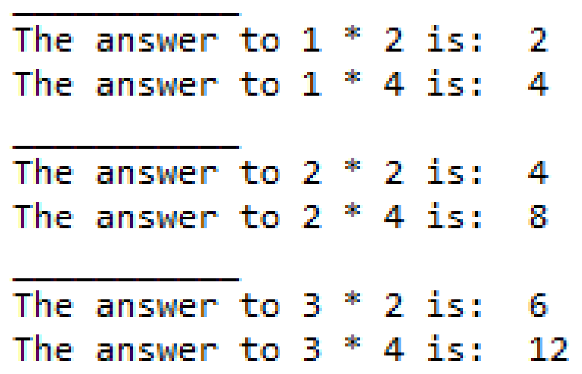
- Create two different lists of integers.
- Using your function, write a nested for loop that cycles through each entries in the first list and multiples it by each of the entries in the second list, and prints the result to the screen.

In:



```
def multiply_function(a, b):
    result = a * b
    return result

numbers_list = [1, 2, 3]
multiplier_list = [2, 4]
for n in numbers_list:
    print("___________")
    for m in multiplier_list:
        current_answer = multiply_function(n, m)
        print("The answer to %d * %d is: " %(n, m), current_answer)
```

Out:

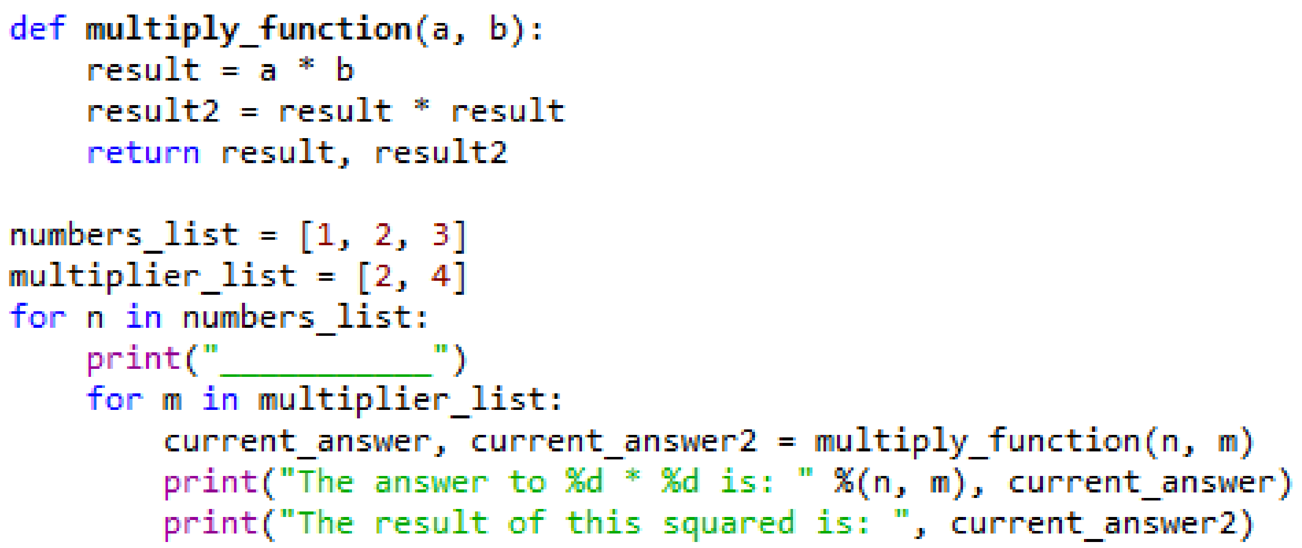


```
___________
The answer to 1 * 2 is:  2
The answer to 1 * 4 is:  4
___________
The answer to 2 * 2 is:  4
The answer to 2 * 4 is:  8
___________
The answer to 3 * 2 is:  6
The answer to 3 * 4 is:  12
```

# Multiple returns

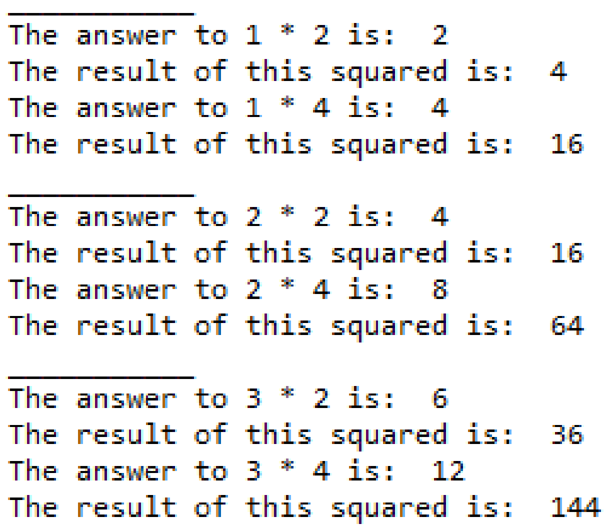
- You can have a function return multiple outputs.

In:



```python
def multiply_function(a, b):
    result = a * b
    result2 = result * result
    return result, result2

numbers_list = [1, 2, 3]
multiplier_list = [2, 4]
for n in numbers_list:
    print("___________")
    for m in multiplier_list:
        current_answer, current_answer2 = multiply_function(n, m)
        print("The answer to %d * %d is: " %(n, m), current_answer)
        print("The result of this squared is: ", current_answer2)
```

Out:



```
___________
The answer to 1 * 2 is:  2
The result of this squared is:  4
The answer to 1 * 4 is:  4
The result of this squared is:  16
___________
The answer to 2 * 2 is:  4
The result of this squared is:  16
The answer to 2 * 4 is:  8
The result of this squared is:  64
___________
The answer to 3 * 2 is:  6
The result of this squared is:  36
The answer to 3 * 4 is:  12
The result of this squared is:  144
```

Any questions?